Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jani Plesničar

Gradnja poligonske mreže proceduralnega drevesa

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

izr. prof. dr. Iztok Lebar Bajec
MENTOR

Ljubljana, 2016

Tematika naloge:

V diplomski nalogi realizirajte sistem proceduralne gradnje dreves. Osredotočite se na gradnjo poligonske mreže. Sistem naj bo dovolj prilagodljiv, da bo omogočal gradnjo verodostojnih in različnih dreves v času izvajanja aplikacije. Rezultate kritično komentirajte.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom izr. prof. dr. Iztoka Lebarja Bajca,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Jani Plesničar, Ljubljana, junij 2016.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jani Plesničar

Gradnja poligonske mreže proceduralnega drevesa

POVZETEK

S pomočjo proceduralnega generiranja objektov lahko zgradimo poligonske mreže objektov, za katere bi pogosto v modelirnih orodjih potrebovali veliko časa. Še posebej takrat, ko bi šlo za objekte, ki morajo biti videti različno, vendar imajo vsi nekaj skupnega. Tipičen primer so drevesa. Ta imajo veje, ki štrlijo stran od debla in se raztezajo proti svetlobi.

V diplomskem delu je predstavljeno zaporedje metod, ki v računalniški grafiki proceduralno generirajo popolnoma proceduralna drevesa, ki jih je nato mogoče uporabiti v računalniških igrah. S pomočjo spremenjenega algoritma kolonizacije prostora (angl. Space Colonization Algorithm) je mogoče zgenerirati skelet drevesa, ki se prilagaja tridimenzionalnemu prostoru, ki je definiran kot oblak točk. Tako zgeneriran skelet drevesa se uporabi za izgradnjo poligonske mreže drevesa. Za ustvarjenje na pogled ustrezne poligonske mreže drevesa potrebujemo kar nekaj korakov. Vsak korak doda k mreži nove podrobnosti. Eden izmed pomembnih korakov je lepljenje cilindrov dreves.

Ključne besede: proceduralni, generator dreves, prostorska kolonizacija, algoritem, konveksna lupina

University of Ljubljana
Faculty of Computer and Information Science

Jani Plesničar

Procedural tree mesh building

ABSTRACT

Procedural generation of objects can be used to build polygonal meshes of objects much faster than with modelling tools. This is particularly true for objects that should look diverse, yet share some common characteristics. Trees are a typical example because their branches grow away from the trunk and extend towards the light.

This thesis presents a sequence of computer graphics methods that enable procedural generation of completely procedural trees that can be used in computer games. The modified Space Colonization Algorithm can help generate the skeleton of the tree that is adapted to the three-dimensional space, which is defined as a point cloud. The skeleton is the foundation for the construction of the polygonal tree mesh. However, a number of steps need to be taken to create an adequate polygonal mesh. Each step adds new details to the mesh and one of the important ones is to merge individual cylinders.

Key words: procedural, tree generator, space colonization, algorithem, convex hull

ZAHVALA

Iskrena hvala mentorju izr. prof. dr. Iztoku Lebarju Bajcu za pomoč, nasvete in strokovnost. Zahvaljujem se tudi moji družini, prijateljem ter partnerici za podporo in številne spodbude.

— Jani Plesničar, Ljubljana, junij 2016.

KAZALO

1 Uvod

V računalniških igrah 3D (3D - 3 dimenzionalno) pogosto uporabimo vizualne trike in modele objektov, ki igralca oz. lik igralca postavijo iz realnega sveta v navidezen prostor. Del fotorealističnih prostorov je teren, vendar je ta brez gozdov pusta pokrajina. Gozdovi dajo igralcu občutek življenja. Vsekakor pa je doživljanje gozdov posameznega igralca odvisno predvsem od njegovih življenjskih izkušenj. Gozdovi so sestavljeni iz posameznih dreves. Zato, da gozd pojmujemo gozd, je ključna prisotnost dreves v veliki skupini. "Gozd je tip kopenskega ekosistema, navzven prepoznaven po poraslosti z gozdnim drevjem."[1] Prav zato se bomo osredotočili na posamezna drevesa, saj so ključna osnova gozdov.

Drevesa v igrah so pogosto enolična, vendar v naravi še zdaleč ni tako. Vsako drevo v naravi je edinstveno in unikatno. V računalniški grafiki se pogosto želimo čim bolj približati kar se le da verodostojni podobi dreves iz narave. Drevesa v računalniški grafiki moramo velikokrat ustvarjati ročno, kar zahteva veliko časa. Na tak način zato lahko ustvarimo le nekaj dreves, ki težko ustvarijo unikatno atmosfero gozdov. Zato je ena

[1] https://sl.wikipedia.org/wiki/Gozd

izmed priročnih rešitev uporaba tako imenovanih proceduralnih algoritmov. Proceduralni algoritmi s pomočjo podatkov o prostoru sami ustvarijo grafično podobo drevesa. Tako je mogoče ustvariti popolnoma unikatna drevesa, posledično gozdove, ne da bi izgubljali čas za ponavljajoče se delo.

Proceduralni algoritmi za gradnjo dreves so sestavljeni iz več zaporednih korakov. Vsak korak je svoj algoritem, ki poskrbi za obdelavo podatkov iz prejšnjega koraka. Vsak korak lahko implementiramo z različnimi pristopi. Vsak ima svoje dobre in slabe lastnosti. Nekateri proceduralni algoritmi za gradnjo dreves potrebujejo še vedno veliko vhodnih podatkov in pravil. Zato algoritmi pogosto uporabljajo nabor šablonskih dreves oz. predhodno sestavljenih dreves, ki jih nato oblikujejo glede na okolico.

V diplomskem delu smo se izognili algoritmom, ki potrebujejo šablone. Drevesa smo gradili s pomočjo sedmih različnih algoritmov. Vsak algoritem je dodal drevesom nove podrobnosti. Teh sedem algoritmov smo razdelili v dve skupini: gradnja skeleta drevesa in gradnja poligonske mreže drevesa. Skelet drevesa da drevesu obliko, poligonska mreža pa vizualno podobo. Za realizacijo algoritmov smo uporabili urejevalnik in grafični pogon Unity 3D.

2 Drevesa

Drevesa za rast kot vsa živa bitja potrebujejo hrano. To pridobivajo s pomočjo funkcije fotosinteze [1]. Svetloba pripomore k predelavi vode in ogljikovega dioksida v formaldehid ter kisik:

$$H_2O + CO_2 \xrightarrow{\text{svetloba}} (CH_2O) + O_2. \tag{2.1}$$

Formaldehid (CH_2O) se nato v zapletenih procesih uporabi za izdelavo sladkorjev, ki jih drevo uporabi za rast. Seveda različne vrste dreves različno dobro predelujejo prej omenjene komponente. Zato imamo na Zemlji toliko različnih vrst dreves z različnimi oblikami krošenj, različnih višin, vrst lesa, oblik listov itd.

Drevesa v naravi rastejo tako, da pridejo do svetlobe po najcenejši poti. Različne vrste dreves rastejo na različne načine. Najbolj opazno razliko vidimo med iglavci in listavci. Za primer vzemimo iglavec smreko, ki ji deblo raste kar se da navpično proti svetlobi in pri tem zaradi majhne površine iglic (listov) zelo na gosto ustvarja veje, ki rastejo skoraj vodoravno, tako da celotno drevo predstavlja pri idealnih pogojih navpični stožec. V gostejših iglastih gozdovih, kjer je borba za svetlobo toliko bolj očitna, opazimo,

da smrekam spodnje veje začnejo odmirati. To se zgodi, ker spodnje veje ne dobivajo dovolj svetlobe.

Seveda svetloba ni edini faktor, ki vpliva na obliko krošnje. Med drugim nanjo vplivajo tudi vlaga, temperatura, veter, vrsta zemlje in fizične ovire (skale, zidovi in druga drevesa) [2]. Obliko krošnje najbolje posnema algoritem za kolonizacijo prostora, katerega delovanje bomo predstavili v poglavju 3.

2.1 Razvoj dreves v računalniški grafiki

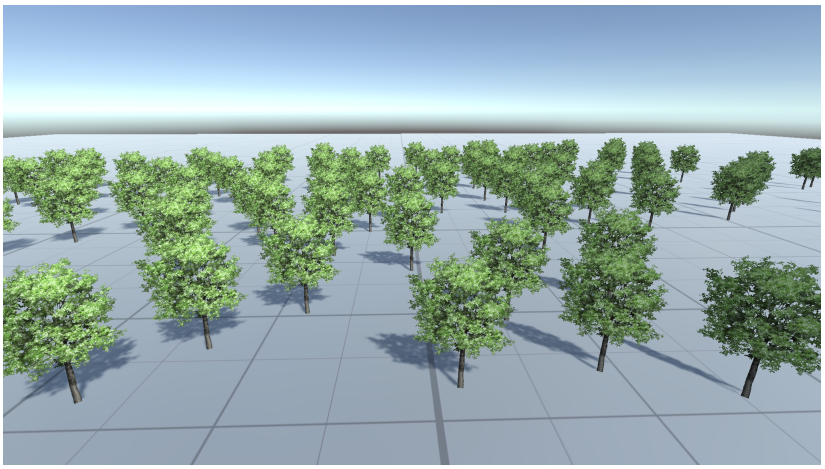
Da lahko danes s pomočjo proceduralnih algoritmov ustvarjamo verodostojna drevesa, se je moralo v teku razvoja računalniške 3D-grafike zgoditi več korakov. Zaradi omejitve strojne opreme so se za imitacijo dreves v preteklosti uporabljali le panoji s teksturami. Panoji so bili vedno obrnjeni v smer kamere. Takšna drevesa so bila na pogled daleč od realne podobe naravnega drevesa. Manjkala jim je globina, bila so enolična in vsa drevesa so izstopala iz okolice. Naslednje izboljšanje v razvoju dreves v računalniški grafiki so bile dodatne stranice panoja, ki so dodale navidezno globino. Vendar so drevesa še vedno bila videti nerealno, še posebej takrat, ko se jim je igralec dovolj približal. Kljub pomanjkljivostim se takšna imitacija dreves še vedno uporablja v računalniški grafiki za prikaz oddaljenih dreves, kjer ni treba izrisovati podrobnosti dreves. Ko je strojna oprema postala dovolj hitra, so se začela v igrah pojavljati drevesa, ki so imela samo debla iz preprostih poligonskih mrež. Kmalu so se pojavila drevesa s kompleksnejšimi poligonskimi mrežami, ki so upodabljala tudi veje. Listi dreves so danes po večini še vedno upodobljeni s pomočjo panojev. Vendar zaradi posebnih tehnik nizanja panojev na veje so drevesa videti dokaj realna. Zato pridejo v poštev proceduralni algoritmi za gradnjo dreves. Glede na vrsto drevesa, prostora in drugih okoljskih značilnosti lahko gradimo skelet dreves.

2.1.1 Drevesa kot preprosti panoji s teksturo

Pano v računalniški grafiki je štirikotnik v 3D-prostoru. Pano ima lastnost, da je vedno obrnjen tako, da gleda proti kameri, ne glede na smer gledanja. Za drevesa ponavadi to ne velja, saj drevesa stojijo (njihovo vrtenje omejeno po navpični osi).

Kot smo že omenili, se je ta tehnika v preteklosti precej uporabljala. Kljub temu, da gre za zastarelo tehniko, jo še vedno srečamo v računalniški grafiki pri izrisovanju skupin listov na geometričnih drevesih [3], šopov trave [4] in oddaljenih dreves [5]. Uporablja

se tudi za izrisovanje dima in raznih delcev prahu ipd. Ker nam ni treba izdelovati 3D-geometrije dreves in potrebujemo le transparentno teksturo, je tehnika izdelave takšnih dreves precej preprosta. Posamezno drevo enostavno narišemo ali pa ga izrežemo iz obdelane fotografije. Ozadje more biti transparentno. Grafični procesor nato izriše teksture dreves na preprosto geometrijo štirikotnika (angl.: Quad), ki je ves čas obrnjen proti igralcu oz. kameri. Tako izrisani elementi velikokrat trpijo za nepravilno izrisanimi prosojnimi deli tekstur, npr. prosojni del drevesa, ki je kameri najbližje, se izriše, kot da za pravokotnikom ni ničesar in na tem mestu nastane luknja v sliki. Do takšnih nepravilnosti pride zaradi prosojnosti, vrstnega reda izrisovanja in globinske slike zaslona; ko štirikotnikov v 3D-prostoru ne razvrščamo glede na razdaljo od kamere in se najprej izriše pravokotnik v ospredju, šele nato pa ostali za njim. Zato je pred izrisovanjem treba razvrstiti vse prosojne pravokotnike oz. trikotnike. Včasih je to bila zelo draga operacija, a danes z modernimi procesorji ne predstavlja več težave. Zaradi enostavnosti implementacije se panoji s teksturo še vedno uporabljajo za šope vej z listi na drevesih.



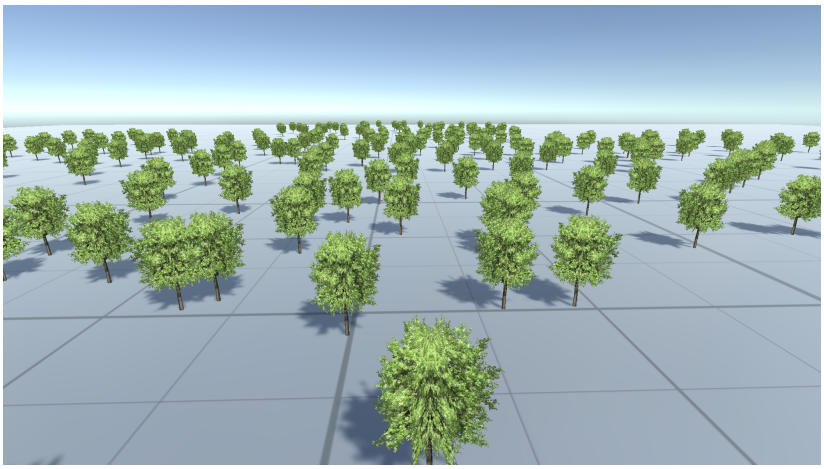
Slika 2.1: Primer dreves s panoji, ki so usmerjeni proti kameri ter omejeni po vertikalni osi.

Drevesa s panoji so zaradi pomanjkanja globinske informacije videti ploščata in jih je pogosto težko izdelati tako, da se vizualno ujemajo s preostalo sceno. Panoji z vertikalno osjo imajo tudi to težavo, da je treba pred vsakim izrisom na zaslon izračunati nove točke na procesorju in jih poslati na grafični procesor. To je še posebej problematična težava na starejših procesorjih, ki ne omogočajo vzporednega izračuna točk. Poleg tega

pošiljanje teh podatkov za vsako sliko zmanjša prepustnost med procesorjem in grafičnim procesorjem za ostale modele v sceni. Panoji brez vertikalne osi te težave nimajo, ker lahko točke štirikotnikov neposredno shranimo v grafični pomnilnik in s pomočjo posebne projekcijske matrike preskočimo preračunavanje na procesorju. Primer dreves s panoji lahko vidimo na sliki 2.1. Drevesa so omejena tako, da se ne vrtijo po vertikalni osi.

2.1.2 Drevesa kot pahljača štirikotnikov

Drevesa lahko posnemamo tudi s pomočjo večjega števila štirikotnikov, sekajočih po sredinski vertikalni osi. Tako lahko zelo enostavno posnemamo globino drevesa. Najenostavnejša verzija takšnega drevesa je sestavljena iz dveh, med seboj pravokotnih štirikotnikov. Ta tehnika nam omogoči, da naložimo poligonsko mrežo dreves v grafični pomnilnik le enkrat. Kompleksnejše verzije dreves imajo lahko več štirikotnikov, nekatere tudi vodoravne.



Slika 2.2: Primer dreves kot pahljača štirikotnikov. Vsako posamezno drevo je sestavljeno iz štirih štirikotnikov.

Takšna drevesa se pogosto uporabljajo v simulacijskih igrah, kjer grafične podrobnosti sveta niso toliko pomembne. Zaradi potrebe po ogromni količini dreves je to najcenejša metoda izrisovanja gozdov, vendar zaradi načina izrisa drevesa so od blizu videti nerealno, a so kljub temu videti bolje kot drevesa na panojih. V igrah, kjer so drevesa del bližnje scene, štirikotniki s teksturami niso dovolj. Zato se velikokrat ta tehnika uporablja kot del sistema več nivojev podrobnosti objekta (angl.: LOD - level of detail),

kjer tehniko uporabimo zgolj za izris oddaljenih dreves. Primer takšnih dreves je na sliki 2.2. Pri najbližjem drevesu lahko opazimo, da ima globino, ki je na sliki 2.1 ne vidimo. Če podrobneje pogledamo, lahko opazimo, da je drevo izrisano iz štirih sekajočih se štirikotnikov. Takšne napake lahko opazimo le od blizu in so v oddaljenih scenah popolnoma neopazne.

2.1.3 Drevesa iz poligonskih mrež

Drevesa iz poligonskih mrež so sestavljena iz večjega števila trikotnikov. Poligonska mreža trikotnikov predstavlja deblo drevesa. Videz drevesa je pogosto odvisen od oblike poligonske mreže in nanesenih tekstur. Listi oz. skupina listov drevesa so pogosto izrisani s pomočjo panojev ali pa z večjim številom prekrivajočih se štirikotnikov. Za pospeševanje izrisa lahko drevesom, ki so bolj oddaljena, odvzamemo nepotrebne šope listov ali pa uporabimo poligonsko mrežo slabše natančnosti z isto ali podobno strukturo.

Za izris realističnega drevesa potrebujemo veliko trikotnikov, torej zelo veliko poligonsko mrežo. Prva drevesa iz poligonskih mrež so bila zaradi omejitve strojne opreme precej primitivna in so imela samo preprosto deblo, narejeno iz nekaj trikotnikov. Veje in listi so bili izrisani s pomočjo štirikotnikov ali preprostih panojev. Ko je strojna oprema začela postajati vedno močnejša, so drevesa začela dobivati tudi kompleksnejša debla. Hkrati so panoje vej začele izpodrivati poligonske mreže vej. Listi oz. skupine listov na vejah so še danes izrisani s pomočjo štirikotnikov zaradi omejitev spomina in grafične moči.

Poligonska drevesa so zaradi enostavnosti izdelovali kar ročno v 3D-modelirnih programih. Ko je drevesna struktura postala zahtevnejša, je bilo kmalu jasno, da hitra izdelava na pogled lepih dreves zahteva preveč ur človeškega dela. Da so drevesa videti skladno z okoljem, je bilo pogosto potrebnih kar nekaj iteracij in prilagajanja dreves okolju. Za izdelavo ogromnega števila različnih dreves se je začelo delati na tako imenovanih proceduralnih tehnikah. Proceduralne tehnike za gradnjo dreves izkoriščajo v svojo prid lastnosti in pravila rasti dreves v naravi; da drevesa rastejo navzgor, da veje rastejo iz debla in listi iz vej. Zaradi abstraktne kompleksnosti gradnje poligonskih mrež dreves gradnja zahteva več zaporednih korakov. Ti koraki zahtevajo, da so prejšnji koraki pravilno zaključeni. To pomeni, da algoritem najprej z določenimi pravili poskrbi za pravilno izdelavo oblike drevesa in nato nanese poligonsko mrežo na skelet drevesa.

Ko je bil ta način izdelave dreves še v povojih, je bila strojna oprema prepočasna, da

bi lahko takšna drevesa v realnem času dodajali v sceno in jih spreminjali. Ponavadi so se drevesa v fazi razvoja gradila kot del statične scene in priložila v končno aplikacijo. Danes to ni več težava, saj so procesorji in grafične kartice dovolj hitri za gradnjo drevesa med nalaganjem scene ali pa jih gradimo celo v času izvajanja aplikacije. Takšna drevesa lahko interaktivno premikamo v sceni in konstantno prilagajamo glede na okolje. Moderne tehnike uporabljajo ogromno vhodnih nastavitev, kar nam omogoči najoptimalnejšo prilagoditev okolju. Da dosežemo, kar se da podoben videz dvojnikom iz narave, takšna moderna orodja posegajo po ogromnih knjižnicah z drevesi, kjer se nato drevesom spreminja samo struktura.

2.2 Obstoječe rešitve

V naslednjih poglavjih bomo predstavili nekaj izmed obstoječih rešitev.

2.2.1 SpeedTree

SpeedTree[1] je obsežno komercialno orodje, ki omogoča interaktivno urejanje dreves v realnem času. Nato lahko na tak način urejeno drevo naključno regeneriramo na različne načine. SpeedTree ima zato veliko zbirko različnih preddefiniranih vrst dreves, ki jih nato s pomočjo pravil L-Sistem [2] spreminja in oblikuje glede na izvorni vzorec drevesa. Drevesa so zato na pogled ustreznejša od dreves, ki jih generiramo v našem diplomskem delu. Med drugim SpeedTree lahko s pomočjo prej omenjene zbirke dreves deluje skoraj popolnoma avtonomno in ustvarja gozdove v igrah. Omogoča animiranje posnemanih dreves v realnem času glede na zunanje sile (veter, udarci ipd.).

2.2.2 Unity: Tree Editor

Unity: Tree Editor[2] je orodje, ki je integrirano v pogonu Unity 3D. Omogoča gradnjo raznovrstnih proceduralnih dreves. Poleg tega nam omogoča tudi neposredno integracijo s samo sceno. Tako lahko določimo vetrovna območja, sile ipd., pogon pa nato sam glede na parametre animira drevesa. Orodje je zato veliko bolj intuitivno v primerjavi z drugimi zunanjimi orodji.

[1] http://www.speedtree.com/
[2] http://docs.unity3d.com/Manual/class-Tree.html

2.2.3 Arbaro

Arbaro[3] je odprtokodna implementacija generatorja dreves [6]. Urejevalnik omogoča le statično generiranje dreves. V urejevalniku moramo torej predpripraviti poligonsko mrežo drevesa, ki se nato uporabi pri sestavljanju scene. Poleg tega Arbaro ne omogoča generiranja skeleta drevesa in je treba celotno drevo ročno sestaviti v urejevalniku.

2.2.4 ngPlant

ngPlant[4] je odprtokodni interaktivni urejevalnik, ki ima med drugim možnost uporabe programske knjižnice v igrah oz. v aplikacijah in generiranju dreves.

2.2.5 Pregled drugih del

Čeprav obstaja že veliko orodij, ki nam omogočajo proceduralno gradnjo dreves, se na tem področju še vedno veliko raziskuje. Oblika generiranih dreves je velikokrat odvisna od vhodnih parametrov, ki jih moramo pogosto nastavljati na roke. Naša metoda generiranja dreves (glej 3. poglavje) potrebuje za vhodne podatke oblak točk, ki predstavljajo krošnjo. V našem primeru generiramo oblak točk s pomočjo enostavnih geometričnih teles. Oblika oblaka točk lahko močno vpliva na videz in verodostojnost celotnega drevesa. Zato je še posebej pomembno, da prilagodimo obliko oblaka s pomočjo dreves iz narave. S pomočjo metode, ki so jo predstavili Argudo in sod. [7], lahko iz ene slike drevesa rekonstruiramo oblak točk, ki ga lahko nato uporabimo v algoritmu za kolonizacijo prostora [8]. Omogoča nam, da lahko iz nekaj slik rekonstruiramo celoten gozd, ki je na pogled lahko videti zelo realno. Ta metoda nam omogoča tudi, da lahko zgradimo več nivojev podrobnosti dreves in tako zmanjšamo obremenitev grafičnega procesorja. Bližnja drevesa lahko izrišemo s pomočjo poligonske mreže, oddaljena pa s pomočjo RDM (angl.: radial distance map).

Algoritma L-Sistem [2] in kolonizacija prostora [8] nista edina načina konstrukcije skeleta drevesa. Proceduralna metoda za nepravilna drevesa (angl.: A procedural method for irregular tree models) [9] nam omogoča, da lahko skelet gradimo tudi s pomočjo iskanja najkrajše poti od vršičkov vej proti skupni točki v grafu. S to metodo lahko gradimo drevesa najrazličnejših vrst in oblik. Pri gradnji skeleta lahko upoštevamo tudi ovire, ki jih je težje implementirati z algoritmom za kolonizacijo prostora. Graf uporablja na

[3] http://arbaro.sourceforge.net/
[4] http://ngplant.org/

poteh rasti uteži. S pomočjo teh lahko simuliramo rast proti svetlobi oz. soncu.

Proceduralni algoritmi pogosto potrebujejo veliko količino vhodnih podatkov. Zato da dosežemo najboljše rezultate je te podatke potrebno pogosto popravljati oz. vnašati na roke. Velikokrat naključno generirani podatki privedejo do nenaravnih rezultatov. Drevesa iz narave najlažje posnemamo s pomočjo algoritmov, ki so najbližje naravnim procesom. Najboljši algoritem, ki tudi posnema naravno evolucijo je genetski algoritem. Algoritem v osnovi ustvari nabor naključnih genskih zapisov. Iz genskih zapisov nato zgradimo skeletno strukturo. Vsak genski zapis vedno generira isto drevo. Drevo lahko nato generiramo s pomočjo L-Sistem algoritma ali prostorske kolonizacije. Tako ustvarjenim drevesom nato določimo oceno. Oceno se lahko izračuna s površino, količino sprejete svetlobe in količino porabljene hrane za rast. Iz nabora ustvarjenih dreves izberemo nekaj najboljših. Genske zapise najboljših dreves lahko nato križamo v en genski zapis z vnašanjem naključnih odsekov zapisa. Ali pa najboljšim drevesom vnesemo nekaj naključnih sprememb v genski zapis ter proces ponovimo, dokler nismo zadovoljni z dobljenimi rezultati. Juš Lozej v svoji diplomski nalogi [10] opiše proces generiranja takšnih dreves. Algoritem oblikuje drevesa glede na svetlobo. Med drugim opiše tudi način gradnje poligonske mreže.

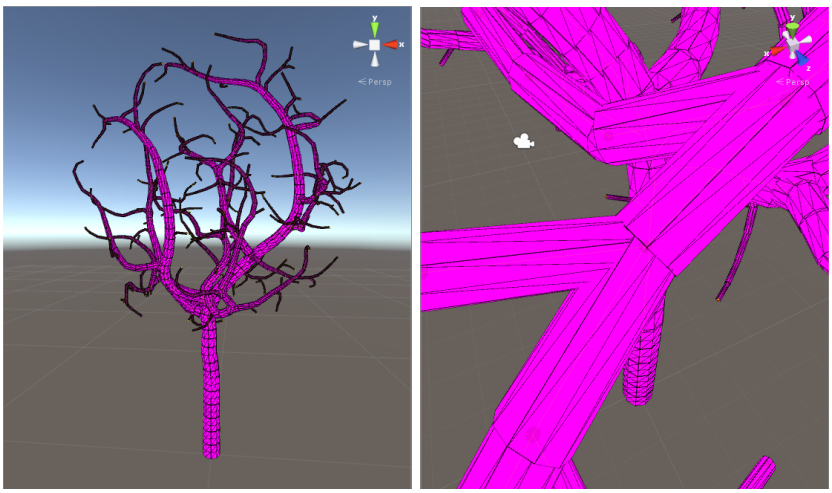
L-Sistem je najbolje uporabljati takrat, ko želimo graditi specifično oblikovana drevesa. Pri takšnih algoritmih se velikokrat zatakne, ko želimo graditi drevesa na katera vplivajo zunanji faktorji, kot so veter, svetloba, ipd. Takšnim sistemom pravimo odprti L-Sistemi, ker so zmožni upoštevati tudi zunanje vplive. Rok Kogovšek s pomočjo generatorja in gramatike [11, str. 27–37] generira naključna drevesa. Drevesa so generirana izven konteksta in neodvisna ena od druge. Hkrati s pomočjo istega generatorja lahko generira gozd dreves. Opisuje tudi, kako lahko s pomočjo vektorjev vpliva [11, str. 27–28] generiramo skelet drevesa.

Ker bi izris celotnega drevesa zahteval precej procesorske in grafične moči se pogosto poslužujemo raznih trikov. Kot smo že omenili lahko krošnjo drevesa izrisujemo s pomočjo panojev. Takšni panoji so za izrisovanje enostavnejši kot poligonska mreža listov. Vendar je proceduralno generiranje takšnih panojev veliko bolj zahtevnejše kot uporaba pred izrisanih skupin vej in listov. V članku [12] je Aleks Jakulin predstavil zaporedje rezanja in mešanja panojev. Tako lahko iz generiranega drevesa dobimo optimizirano drevo, katero je sestavljeno iz poligonske mreže debla in skrajšanih vej ter narezanih slik krošnje.

3 Pristop h gradnji drevesa

Če narišemo drevesa iz narave na list papirja samo s črtami, opazimo, da imajo drevesa pravzaprav preprosto strukturo skeleta, kjer najbolj preprosta drevesa na pogled izhajajo iz tal iz ene točke in se nato vejijo tako, da kar se da na široko raztezajo veje. Zato je prvi korak pri gradnji poligonske mreže gradnja skeleta drevesa, ki ga oblečemo s poligonsko mrežo. Skelet drevesa lahko najbolj enostavno gradimo s pomočjo dveh algoritmov. To sta algoritem L-Sistem [2], kjer za vhodne podatke potrebujemo kompleksna pravila, in algoritem za kolonizacijo 3D-prostora [8]. Tako dobimo približno obliko drevesa, nato je treba ustvariti poligonsko mrežo drevesa. Tega se lahko lotimo na nekaj načinov. Najenostavnejši način je, da na skelet drevesa preprosto nanesemo cilindre, glej sliko 3.1, in kjer ima drevo stičišča, cilindre preprosto podaljšamo tako, da se med seboj prekrivajo.

Vendar je to pogosto videti precej nerealno zaradi ostrih oz. prisekanih prehodov med vejami. To popravimo tako, da stičišča vej zgradimo s pomočjo kakšnega drugega algoritma, kot je na primer algoritem za gradnjo konveksnih lupin [13], ali pa z uporabo konstruktivne polne geometrije (angl.: constructive solid geometry) [14], kjer z Boolovimi operacijami združimo cilindre v eno celoto. Krošnjo drevesa nato zgradimo s pomočjo



Slika 3.1: Rezultat naivne gradnje cilindrov in prikaz pomanjkljivosti.

naključno postavljenih štirikotnikov.

Gradnja drevesa zahteva korake: Gradnja skeleta drevesa → Priprava na oblačenje s poligonsko mrežo → Združevanje cilindrov → Deljenje poligonske mreže → Nanašanje tekstur na poligonsko mrežo → Gradnja listnate krošnje

3.1 Gradnja skeleta drevesa

Za gradnjo skeleta drevesa smo uporabili algoritem za kolonizacijo prostora [8]. Algoritem temelji na oblaku točk, ki se razprostira po 3D-prostoru, vsaka točka predstavlja atraktor. Smer rasti oz. vejenja je odvisna od gostote točk oz. atraktorjev v določenem predelu oblaka točk. Algoritem posnema kolonizacijo prostora. Najlažje si lahko predstavljamo to, kot naivno kolonizacijo vesolja, kjer so atraktorji planeti, ki jih želimo naseliti, vendar lahko to naredimo le z omejenimi razdaljami skokov med planeti. Zato je možno, da nekaterih planetov, ki so predaleč, nikoli ne naselimo.

Takšno analogijo lahko prenesemo tudi na drevesa. Iz prejšnjega poglavja vemo, da drevesa, da živijo, potrebujejo prostor za svoje krošnje. Ta prostor lahko zapolnimo na enak način. Vendar so v tem primeru atraktorji še nezaseden prostor oz. prostor s svetlobo. Kolonizacijski algoritem v tem primeru začne pri tleh.

Enakomerno porazdeljene točke predstavljajo uravnoteženo krošnjo. Gradnja skeleta drevesa se začne z gradnjo debla. V drugi fazi algoritem gradi veje in v tej fazi višina

debla močno vpliva na obliko same krošnje. Če je deblo visoko, bomo dobili drevo, ki je bolj podobno smreki. Če pa je deblo nizko, bo krošnja bila bolj podobna grmu. Algoritem za kolonizacijo prostora je deterministično, kar pomeni, da bo iz istega nabora točk vsakič ustvaril isti skelet drevesa.

3.2 Priprava na oblačenje s poligonsko mrežo

V naravi drevesa in njihova debla ter veje niso samo črte. Zato smo vsakemu odseku para (otrok-starš) dodali podatek o volumnu oz. debelini. Geometrijsko vsak odsek predstavlja prisekan stožec. Premer večje osnovne ploskve je premer starša, medtem ko je premer manjše osnovne ploskve debelina trenutnega odseka. Višina prisekanega stožca je dolžina odseka (otroka).

Pred tem smo morali najprej izračunati debelino posameznega odseka. Debelino smo definirali kot polmer odseka. Debeline odsekov smo izračunali tako, da smo končnim odsekom (abstraktno listom) priredili začetno debelino. Nato smo s pomočjo sklada potovali od listov proti korenu drevesa in vsakemu odseku dodelili vrednost debeline otroka ter prišteli dodatno vrednost. Ko smo naleteli na odseke, ki so imeli več kot enega otroka, smo za trenutni odsek vzeli debelino debelejšega otroka.

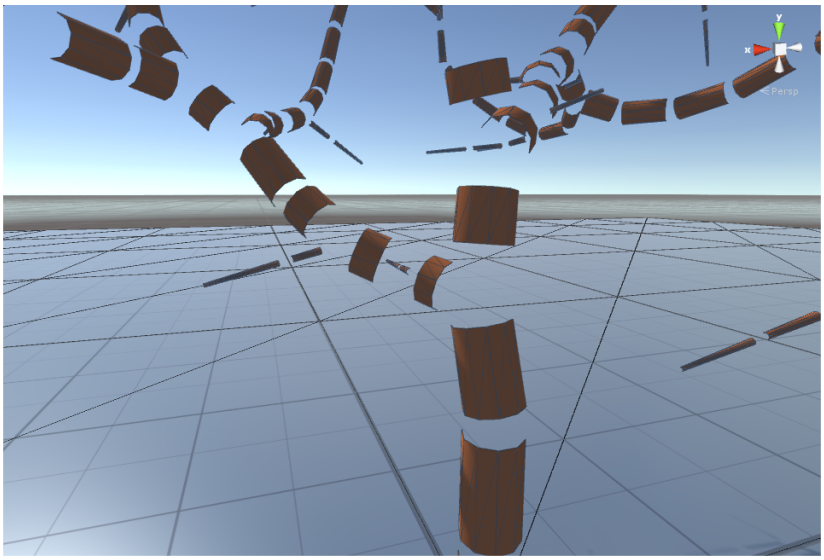
Ker so daljši odseki akumulirali več debeline kot krajši odseki, bo zaradi tega drevo videti, kakor da ima debelejše glavne in tanjše stranske veje.

Če bi cilindre postavili na dele veje in debla, bi hitro opazili, da se cilindri med seboj prekrivajo. Zato smo morali izračunati tudi odmike začetka in konca cilindra, kar pomeni, da smo onemogočili prekrivanja na spojih cilindrov. To smo naredili po metodi, imenovani prirezanovanje vej (angl.: Branch Trimming), omenjeni v [15, str. 26–28]. Metoda ima pet različnih primerov, ki jih je treba vse preveriti. Ker metoda ni procesorsko zahtevna, se izvede izredno hitro, zato lahko ta postopek izvedemo za vsak par spoja brez večjih obremenitev.

3.3 Gradnja poligonske mreže

Iz skeleta drevesa in podatkov o debelini odsekov smo lahko začeli z gradnjo poligonske mreže. Ta je nekoliko bolj zapletena od gradnje skeleta in izračuna posameznih premerov odsekov. Izvesti je treba več različnih korakov, kjer vsak končni poligonski mreži drevesa doda nove podrobnosti.

Če pogledamo, kako narava ustvarja nosilne strukture, opazimo, da je veliko struktur pogosto sestavljenih iz cilindrov. Ker so cilindri fizično zelo močni in omogočajo velike nosilnosti, imajo vsa drevesa debla ter veje v obliki cilindrov. Zato lahko varno rečemo, da so veje in debla v osnovi cilindraste oblike. Cilinder je 3D-telo, ki je v računalniški grafiki pogosto predstavljen s prizmo, ki ima za osnovno ploskev n-kotnik s 6 ali več koti. Poligonsko mrežo začnemo graditi tako, da najprej postavimo na skelet drevesa

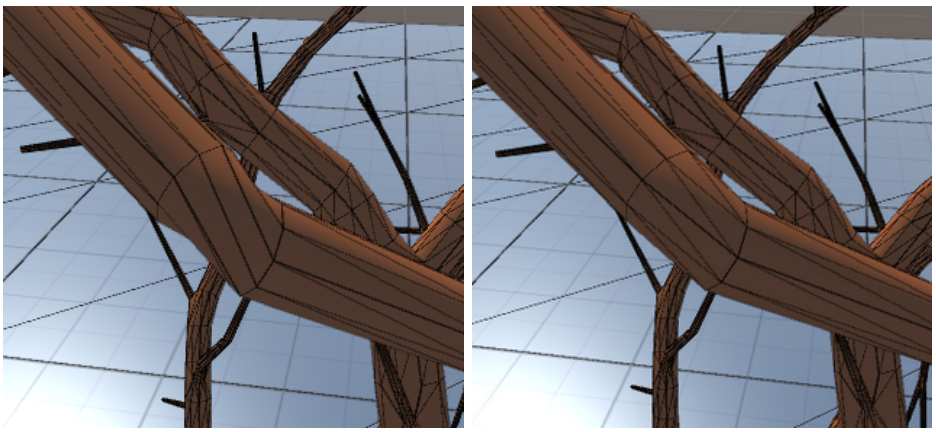


Slika 3.2: Skrajšani cilindri, pripravljeni na šivanje poligonske mreže z uporabo metode konveksne lupine.

kratke cilindre brez osnovnih ploskev. Premer cilindrov je odvisen od koraka pri izračunu debelin. Cilindre smo skrajšali (glej sliko 3.2) glede na izračunane odmike v prejšnjem koraku.

3.3.1 Združevanje cilindrov

Ko smo zaključili s korakom gradnje poligonske mreže, smo lahko združili cilindre na dva načina. Združevanje cilindrov z enim otrokom je bilo najenostavnejše. Te odseke smo gradili po principu, ki je na videz podoben tistemu, ki smo ga uporabili za gradnjo cilindrov. Zaradi funkcije, s katero izračunamo vektor v ravnini, ki je pravokotna na smer rasti, se lahko zgodi nepravilna sučna poravnava, kar lahko vidimo na sliki 3.3a. Zato smo morali poiskati točko cilindra, ki je najbližja točki na naslednjem cilindru. Ko



(a) (b)

Slika 3.3: (a) Naiven spoj; (b) Pravilen spoj.

smo to točko našli, smo lahko cilindra pravilno povezali. Če tega ne bi naredili, bi nam nepravilno zasukani cilindri delali težave s teksturami in z osvetlitvijo. Tako pa smo dobili pravilen spoj, ki ga vidimo na sliki 3.3b.

Združevanje spojev z več otroki je nekoliko bolj zahtevno, saj zahteva, da povežemo sosednje otroke. Za to nalogo smo izbrali metodo, ki je opisana v [15, str. 40–42]. Metoda izkorišča lastnost krogle, ki je vedno konveksne oblike. Na to kroglo lahko postavimo krožnice, ki se ne smejo stikati – za to je poskrbel algoritem za prirezovanje cilindrov. Ker je krogla v osnovi konveksne oblike, lahko brez zadržkov rečemo, da vse točke krožnic na krogli sestavljajo plašč konveksne lupine. In zato so povezave med točkami neprekrivajoče. Za ta korak gradnje spoja smo uporabili algoritem za gradnjo konveksnih lupin, imenovan QuickHull [13]. Algoritem QuickHull je eden izmed bolj enostavnih algoritmov za gradnjo konveksne lupine. Hkrati ima tudi to lastnost, da z uporabo primernih podatkovnih struktur zagotavlja hitro gradnjo konveksnih lupin. Algoritem smo za naš namen bili primorani nekoliko dodelati, saj je za vhodne podatke moral sprejemati indekse točk namesto samih točk. To nam je olajšalo kasnejše povezovanje cilindrov.

3.3.2 Deljenje poligonske mreže

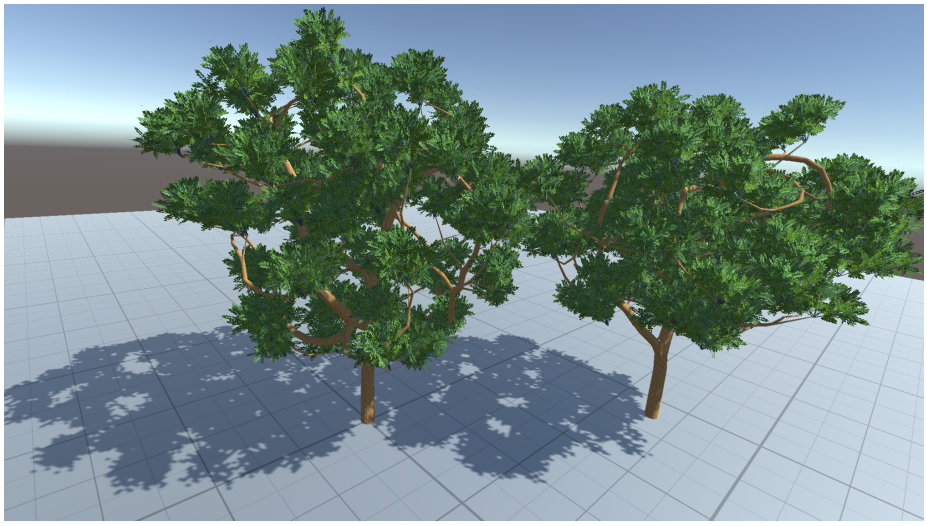
Zaradi sestavljanja poligonske mreže drevesa iz primitivnih teles in konveksne lupine je poligonska mreža ponekod groba, zato smo poligonski mreži morali povečati resolucijo. To smo naredili s pomočjo algoritma za delitev trikotnikov [16]. Rezultat je gostejša

mreža, ki je na pogled videti bolj podrobna.

3.3.3 Nanašanje tekstur na poligonsko mrežo

Sama poligonska mreža brez tekstur je videti dolgočasno in brez življenja, zato smo morali pri gradnji poligonske mreže upoštevati tudi UV-koordinate. UV-preslikovanje je proces preslikave 2D-slike na 3D-objekt. To naredimo tako, da vsaki točki na poligonski mreži dodamo U- in V-vrednost, ki predstavljata še koordinato v 2D-prostoru. Tako lahko sliko preslikamo z 2D- ravnine v 3D-prostor.

3.4 Gradnja listnate krošnje



Slika 3.4: Končana krošnja.

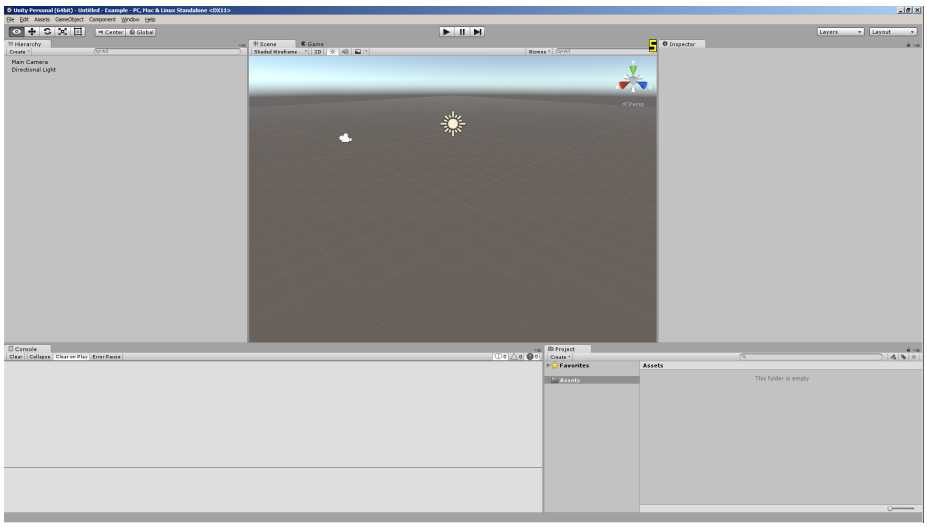
Da smo se lahko približali videzu dreves iz narave, smo morali drevesom dodati tudi liste. Pri tem smo uporabili preverjeno tehniko z uporabo treh sekajočih se štirikotnikov in krošnje listov, izrisanih na eno prosojno teksturo. Tako smo dobili osnovni šop listov, ki smo ga z uporabo skripte klonirali čez celotno drevo. Osnovni šopi so se nato združili v novo mrežo. Postavitev šopov je bila odvisna od tega, kje so se veje na drevesu zaključile. Orientacija listov je naključna. Rezultat vidimo na sliki 3.4.

4 Implementacija

Unity 3D (na sliki 4.1) je orodje za razvijanje 3D-aplikacij oz. 3D-iger. Med drugim je mogoče s tem orodjem razvijati tudi 2D-aplikacije ali 2D-igre. Uporablja se za razvoj na platformah, kot so PC, konzole in mobilne platforme Android, Windows OS ter iOS. Omogoča nam hiter razvoj in zaradi interaktivnega urejevalnika enostavno urejanje scen. Pri tem lahko dodatno po potrebi ustvarjamo razširitve za urejevalnik. Za razvoj dodatnih komponent in razširitev lahko uporabimo tri različne jezike, in sicer Visual C#, Unity Script in JavaScript.

Unity 3D za pozicioniranje objektov uporablja scenski graf (ang.: Scene Graph), kjer vsako vozlišče predstavlja transformacijo otrok glede na starše v drevesnem grafu. Vsaka transformacija oz. vozlišče drevesa ima zapisan lokalni položaj, ki je definiran kot vektor treh komponent x, y, z, in lokalno rotacijo v obliki kvaterniona (angl.: Quaternion) glede na starša. Vsako vozlišče ima lahko svoje ime, ki ni nujno, da je enolično. Več vozlišč si lahko deli isto ime.

Skriptne komponente v Unity 3D-pogonu dedujejo iz MonoBehaviour razreda. Ta razred deluje kot lepilo med notranjimi podatkovnimi strukturami in vrhnjim skriptnim



Slika 4.1: Unity 3D - Glavno okno.

jezikom ter tako omogoči, da lahko dedujoče razrede pripnemo na vozlišča scenskega grafa. Tako lahko upravljamo direktno z vozlišči v sceni. S pogonom Unity izrišemo poligonsko mrežo tako, da na poljubno vozlišče v sceni pripnemo dve že vgrajeni komponenti, imenovani MeshFilter in MeshRenderer.

Ti dve komponenti nam že sam Unity 3D ponuja za osnovno izgradnjo scene. MeshRenderer je izrisovalec poligonske mreže, ki preda informacije o poligonski mreži in o materialu notranjemu grafičnmu upravljalniku, ki deluje popolnoma samostojno ter je uporabniku urejevalnika skrit in dostopen le preko nizkonivojskih ukazov, do katerih lahko dostopamo samo z jeziki, kot sta C in C++. Izrisovalec poligonske mreže sam poskrbi za združevanje objektov v isto serijo (angl.: batch) istih materialov. Grafični upravljalnik nato sam poskrbi za izbiro pravega materiala, tekstur in senčilnika. MeshFilter ali filter poligonske mreže poskrbi za to, da se poligonska mreža naloži iz vira in nato preda izrisovalcu poligonske mreže. Objekt s poligonsko predstavitvijo ali poligonsko mrežo ima tabelo 3D-točk, ki predstavljajo oglišča trikotnikov, tabelo indeksov točk, ki zaključujejo trikotnike, nabor petih tabel za UV-koordinate za nanašanje tekstur, tabelo normal za senčenje ter tabelo tangent za reliefno senčenje.

Za implementacijo generatorja smo uporabili programski jezik C#, ki je najhitrejši izmed programskih jezikov na voljo in ima nekaj dodatnih lastnosti.

S pomočjo Unity 3D-urejevalnika smo ustvarili testno sceno, v kateri smo postavili

drevesno strukturo, kjer vozlišča predstavljajo odseke – dele vej in debla. Vsi odseki imajo pripete skriptne komponente. Skriptno komponento oz. razred smo poimenovali TreePart in ima v sebi nekaj lokalnih spremenljivk.

Mrežo poligonov smo vsakič posodobili preko posebne razširitve, narejene za to nalogo. Poligonska mreža se je s pomočjo razširitve urjevalnika posodabljala na komponentah MeshFilter in MeshRenderer. MeshFilter in MeshRenderer smo pripeli na glavno vozlišče v scenskemu grafu.

Za boljše izkoristke pri izrisovanju smo celotno drevo spravili le v eno poligonsko mrežo, vendar je to s seboj prineslo nekaj težav, kot je na primer omejitev števila točk poligonske mreže na ne predznačeno 16-bitno vrednost ($65534 = 2^{16} - 2$). To je omejitev grafičnega pogona Unity 3D.

Posebno vozlišče smo uporabili kot vsebovalnik oblaka točk, ki je predstavljal obliko krošnje. Na vsako točko smo preko skriptnega urejevalnika scen avtomatsko pripeli komponento SpacePoint. Ta komponenta nam je kasneje prišla prav pri gradnji abstraktnega drevesa. Točke smo lahko vsakič poljubno regenerirali.

4.1 Podrobnejši opisi algoritmov

V naslednjih poglavjih bomo podrobneje opisali najbolj bistvene izmed uporabljenih algoritmov.

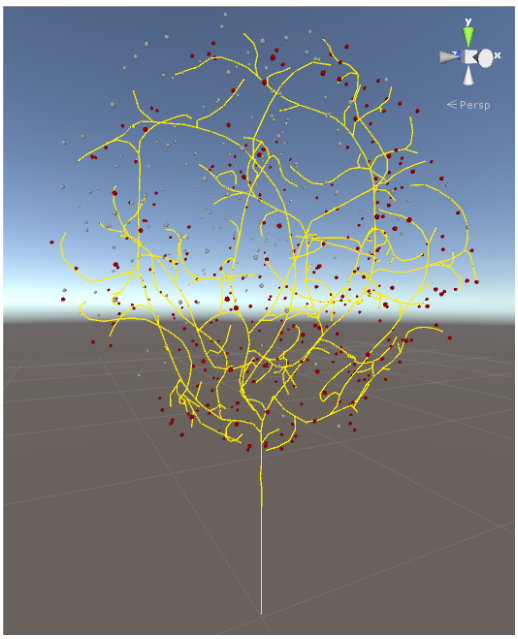
4.1.1 Kolonizacija prostora

Osnovni algoritem za kolonizacijo prostora [8] ima nekaj težav pri ustvarjanju nepotrebnih duplikatnih odsekov. Zato smo pri ustvarjanju novih odsekov dodali dva dodatna pogoja, ki odpravita duplikate. V nekaterih primerih se lahko zgodi, da so zavoji vej ostri. Takšne veje iznakažejo videz drevesa in jih je treba nekako popraviti. To smo storili z dodatnim izračunom, ki premakne starša vej na povprečno lokacijo.

Izvorni algoritem ima v osnovi dve fazi. Prva faza poskrbi za postavitev debla, druga faza poskrbi za rast vej. Druga faza se je izvajala, dokler nismo ustvarili dovolj vej ali pa nam ni zmanjkalo točk.

Faza gradnje debla

Skelet debla smo začeli graditi v točki, imenovani koren. Ustvarili smo toliko enako dolgih zaporednih odsekov debla, dokler nismo dosegli maksimalne višine debla. Smer rasti je



Slika 4.2: Skelet drevesa po dvajsetih iteracija algoritma.

bila določena s preddefinirano smerjo 3D-vektorja.

Faza gradnje skeleta vej

Veje smo gradili iterativno. Vsaka iteracija je dodala drevesu nov nabor vej in izločila atraktorje, ki so že vplivali na rast drevesa. Vsaka iteracija je tudi povečala maksimalno globino drevesa za ena.

V vsaki iteraciji smo za vsak neizločen atraktor poiskali veje, na katere lahko atraktor vpliva[1]. Med drugim smo morali poiskati tudi najbližjo vejo atraktorju. Če je bila veja bližje atraktorju kot minimalna razdalja območja vpliva, smo atraktor izločili in nadaljevali z drugimi.

Za vsako vejo, ki se je tako znašla v območju neizločenih atraktorjev, smo lahko izračunali povprečen vektor relativne smeri do vseh atraktorjev. Tako dobljeni vektor smo nato uporabili za ustvarjanje novih vej, katerih dolžino smo podali kot vhodni parameter. Za vsako vejo smo lahko ustvarili maksimalno dve novi veji. Če algoritem oz. veje ne dosežejo vseh atraktorjev, ti nikoli ne vplivajo na obliko drevesa.

Algoritmu smo dodali tudi pravilo, ki je v posebnih primerih izločalo dvojnike že

[1] Območje vpliva je minimalna in maksimalna preddefinirana razdalja med vejo in atraktorjem.

obstoječih vej. Zato lahko v koraku krajšanja cilindrov predpostavljamo, da se sosednje veje nikoli ne prekrivajo.

Iterativno fazo smo ponavljali, dokler nismo prišli do želene razvejanosti drevsa oz. dokler nam ni zmanjkalo neobiskanih atraktorjev. Na sliki 4.2 smo algoritem iterirali 20-krat. Obiskani atraktorji so rdeče barve, neobiskani pa sive. Rumene črte na sliki 4.2 predstavljajo abstraktno drevo oz. skelet drevesa, na katerega smo kasneje nanesli poligonsko mrežo.

4.1.2 Krajšanje cilindrov

Izračunati je bilo treba tudi odmike cilindrov. To smo naredili po metodi, imenovani prirezanovanje vej, omenjeni v [15, str. 26–28]. Da smo lahko pravilno izračunali odmik cilindrov, smo morali poznati kot med spojenimi vejami in njihovo debelino, izračunano v prejšnjem koraku. Kote izračunamo s pomočjo skalarnega produkta kombinacij vej in starša oz. odsekov. Po izvorni metodi moramo upoštevati pet različnih možnih izidov, vendar, ker v naši implementaciji predpostavljamo, da se veje za kolonizacijo prostora nikoli ne stikajo, lahko varno rečemo, da moramo pokriti samo tri primere:

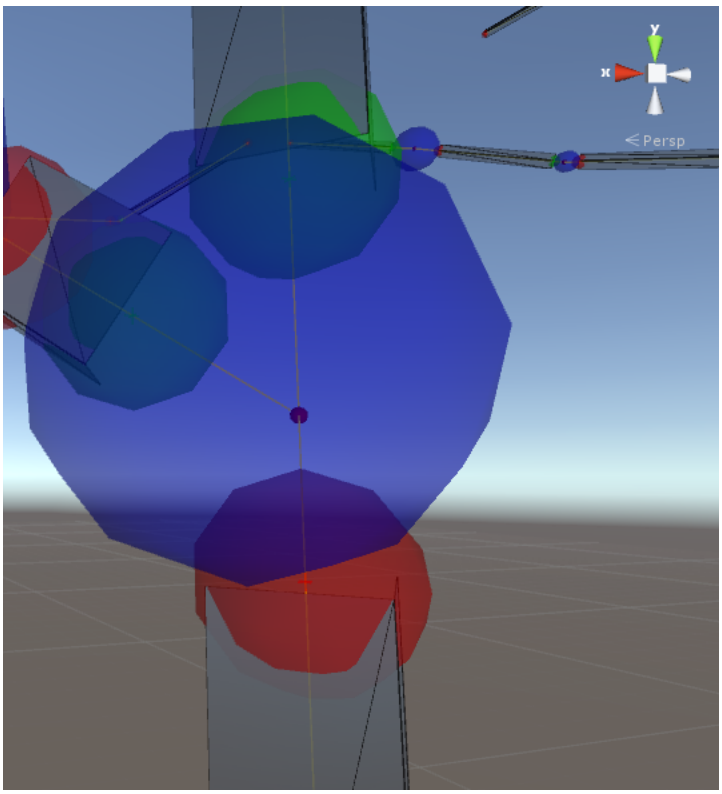
- skalarni produkt znaša 0 takrat, ko sta si dva odseka pravokotna,
- skalarni produkt znaša -1 takrat, ko sta si dva odseka vzporedna, vendar kažeta v obratni smeri drug drugemu (primer: otrok in popolnoma vzporedni trenutni odsek),
- če je znašal skalarni produkt med -1 in 0 ali med 0 in 1, smo najprej izračunali kot α iz skalarnega produkta s pomočjo inverza kosinus funkcije. Nato je bilo treba izračunati odmik za prvega v paru:

$$offset_A = \frac{radius_A}{\tan\alpha} + \frac{radius_B}{\sin\alpha}. \tag{4.1}$$

Iz tako dobljenega odmika smo nato s pomočjo Pitagorovega izreka izračunali odmik drugega v paru:

$$offset_B = \sqrt{thickness_A^2 + thickness_B^2 + offset_A^2}. \tag{4.2}$$

Prvemu in drugemu otroku v paru smo vedno nastavili nov odmik takrat, ko je ta bil večji od odmika prejšnjega izračuna. Tako smo se izognili možnemu premajhnemu odmiku, ko ima odsek več kot dva otroka. Končni odmik smo izračunali tako, da smo



Slika 4.3: Zamik cilindrov ter krogla, ki predstavlja center povezav.

dolžino odseka množili s preddefinirano vrednostjo 0.8. Če je to bil zadnji odsek (list), smo končni odmik nastavili kot dolžino odseka. Primer odmikov lahko vidimo na sliki 4.3. Zelena kroga predstavljata začetka, rdeč krog pa konec cilindra. Modra krogla predstavlja sredino teh treh.

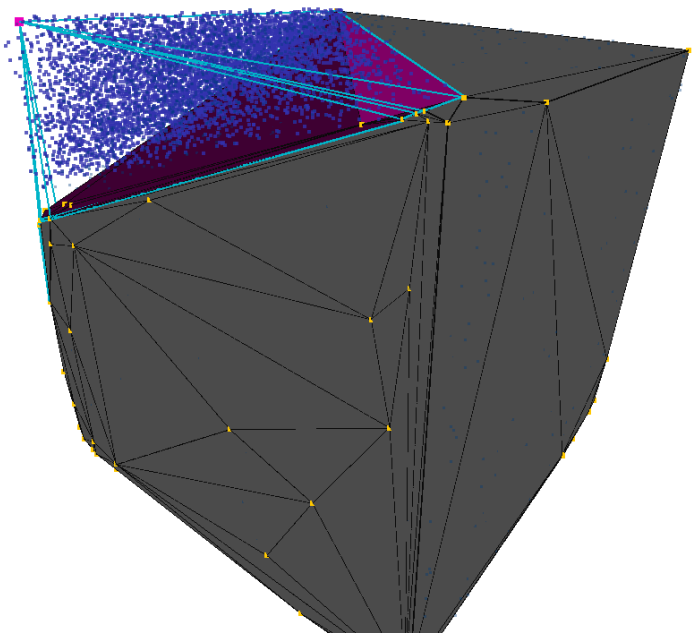
4.1.3 Gradnja poligonske mreže

Poligonsko mrežo odsekov smo začeli graditi pri korenini. To smo naredili s pomočjo iteracije vrste (angl.: Queue), kar nam je omogočilo iteracijo po širini (algoritem potuje čez vse otroke na isti globini, nato pa nadaljuje globlje – znan pod angleškim imenom breath-first iteration). Ker je kos veje oz. debla le črta od predhodnika do naslednika, smo lahko določili smer cilindra.

Poligonsko mrežo cilindrov smo zgradili s pomočjo vrtenja točk okrog osi veje oz. debla, tako da je na začetku in koncu nastal obroč točk. Obroč točk na začetku cilindra

je debelejši kot obroč na koncu. Točke smo nato s pomočjo indeksov povezali v trikotnike, ki jih grafični pogon lahko izriše.

4.1.4 Algoritem za gradnjo konveksne lupine



Slika 4.4: QuickHull korak gradnje konveksne lupine. (Vir: Thomas Diewald)[2]

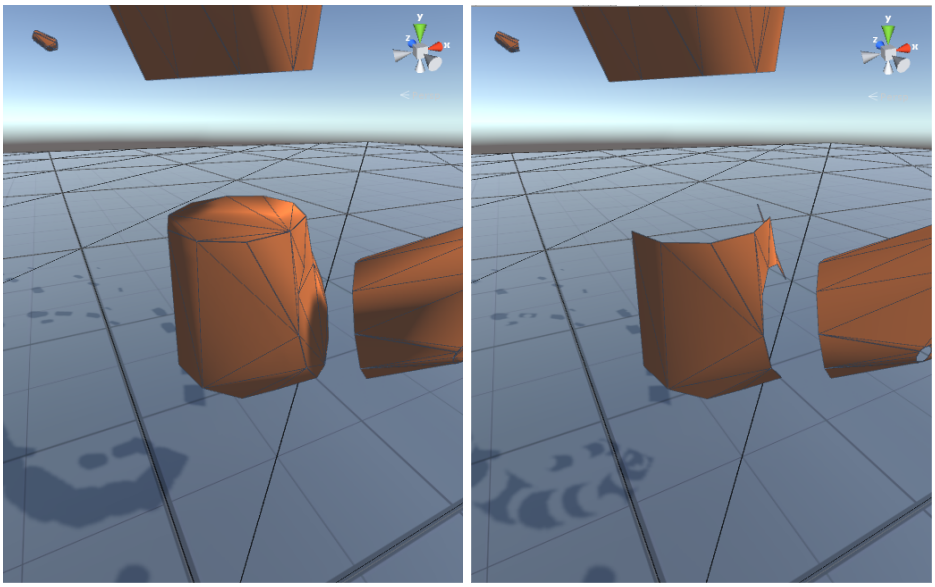
Konveksna lupina ali ogrinjača[3] množiče točk X v realnem vektorskem prostoru V je v matematiki najmanjša konveksna množica, ki vsebuje X kot podmnožico. Drugače povedano, konveksna lupina predstavlja poligonsko mrežo oz. ogrinjalo, ki popolnoma objame množico točk.

Algoritmi za gradnjo konveksnih lupin so v dveh dimenzijah bolj enostavni od algoritmov v treh dimenzijah. V dveh dimenzijah imamo namreč vse točke na eni ploskvi in pri tem iščemo samo najbolj ekstremne točke, ki jih lahko enostavno zaporedoma povezujemo. V treh dimenzijah pa moramo poleg iskanja najbolj ekstremne točke iskati tudi najbližje okoliške točke in način povezave okoliških točk. Konveksne lupine se precej

[2] http://goo.gl/ACUKxk
[3] https://goo.gl/bTV7Bb

uporabljajo v fizikalnih pogonih za detekcijo trkov in za izračun približnega volumna objektov.

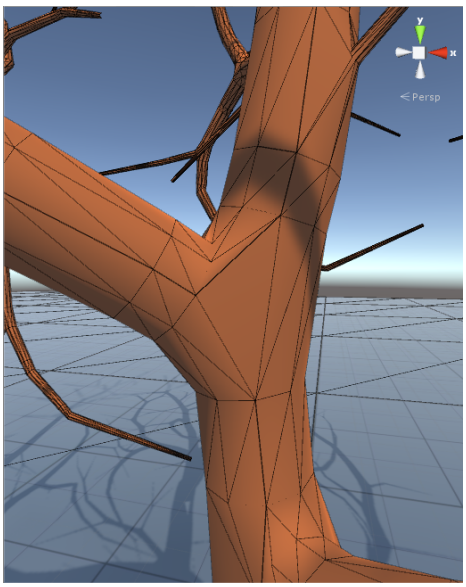
Algoritem QuickHull izkorišča preprosti princip 3D-simpleksa. Za pravilno delovanje potrebujmo za vhodne podatke vsaj 4 točke, kar je logično, če želimo dobiti geometrično telo.



(a) (b)

Slika 4.5: (a) Rezultat algoritma QuickHull algoritma; (b) Odstranjene nevidne ploskve.

Algoritem QuickHull ima začetno in iteracijsko fazo. Algoritem v prvi fazi poišče v oblaku točk 4 ekstremne točke, ki definirajo 4 ploskve 3D-simpleks telesa, imenovanega tetraeder. V drugi fazi se razvrstijo preostale točke med te 4 ploskve. Točka lahko pripada samo eni ali več ploskvam. Da pripada neki ploskvi, mora točka osvetljevati ploskve. Na sliki 4.4 vijolično obarvana točka osvetljuje dve ploskvi oz. trikotnika, ki sta obarvana vijolično. Če točka ne pripada nobeni ploskvi, lahko predpostavimo, da je že znotraj lupine. V vsaki iteraciji algoritma poiščemo najbolj oddaljeno točko, ki pripada trenutni ploskvi. Točko povežemo z oglišči vidnih ploskev in tako ustvarimo nove ploskve oz. trikotnike. Preostale točke na isti način razvrstimo med ploskve in nato izbrišemo staro ploskev. To ponavljamo, dokler nam ne zmanjka ploskev s točkami. Algoritem nam vrne trikotnike oz. ploskve konveksne lupine.



Slika 4.6: Zaključen spoj odseka dveh otrok.

Tako dobljeno konveksno telo (Slika 4.5a) ima to težavo, da ima dele, ki se povezujejo s sosednjimi cilindri zaprte. Te ploskve bi lahko pustili nedotaknjene, ampak ker jih ob normalnem izrisu ne vidimo, jih lahko odstranimo s primerjanjem indeksov točk. Tako dobimo odprt spoj (Slika 4.5b). Poligonsko mrežo spojev cilindrov enostavno združimo v isto poligonsko mrežo. Poligonska mreža drevesa je tako končana (Slika 4.6).

4.1.5 Deljenje mreže

V naši implementaciji smo za deljene poligonske mreže uporabili algoritem deljenja mreže, ki je predstavljen na wiki straneh Unity 3D[4]. Implementacijo algoritma smo naredili tako, da smo lahko spreminjali relief novo nastalih točk v poligonski mreži. Tako smo dobili bolj naravne krivulje. Glej algoritem 1.

Vsakemu trikotniku smo na vse tri robove postavili nove točke, ki smo jih povezali tako, da smo dobili 4 nove trikotnike, star trikotnik pa odstranili. Rob zato predstavlja par prvega in drugega indeksa starega trikotnika. Za ohranitev deljenja novih točk med trikotniki (za predstavljanje povezane poligonske mreže indeksov) smo morali robove shraniti v preslikave robov starega trikotnika v indeks nove točke. Tako smo preprečili ustvarjanje duplikatnih točk.

[4] http://wiki.unity3d.com/index.php?title=MeshHelper

Algoritem 1: Procedura za izračun točke robov.

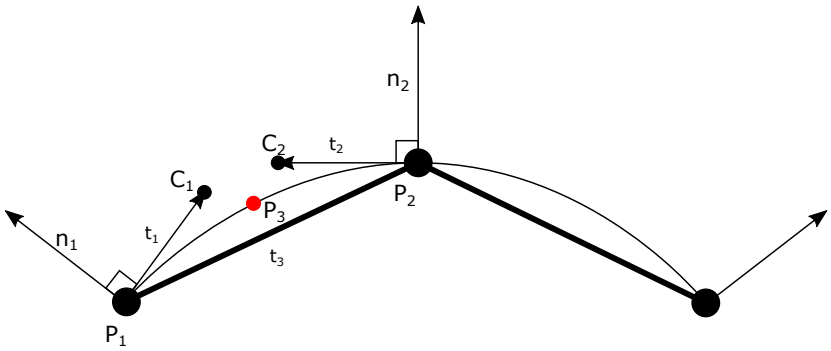
procedura CreateVertex $(index1, index2, tocke, normale)$;

Vhod: Dve pozitivni celi števili $index1$ in $index2$ ter tabeli $tocke$ in $normale$

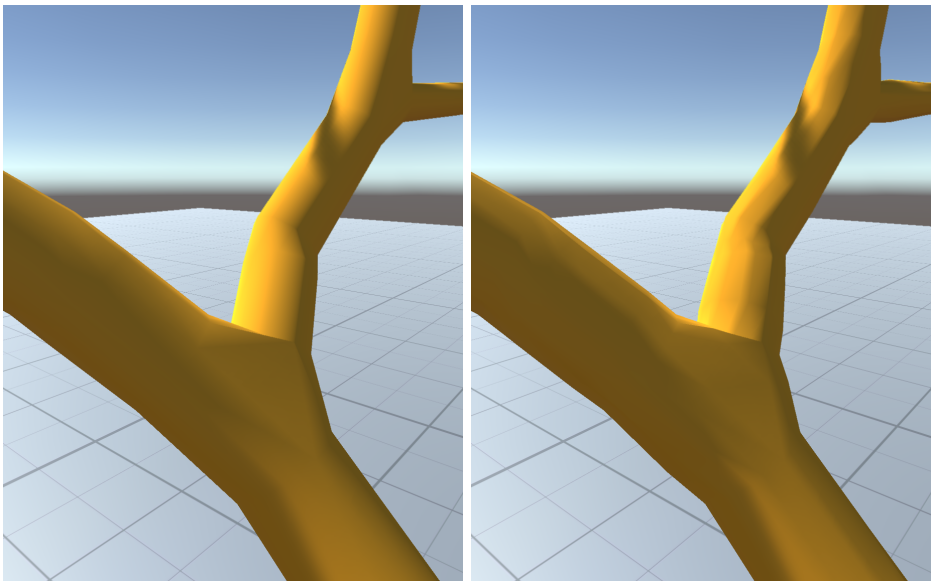
$p0 \leftarrow tocke[index1]$;
$p1 \leftarrow tocke[index2]$;
$n0 \leftarrow normale[index1]$;
$n1 \leftarrow normale[index2]$;
$dotP \leftarrow n0 \cdot n1$;
if $dotP = 0 \vee dotP = 1 \vee dotP = -1$ **then**
    $tocke \leftarrow tocke + (p0 + p1) * 0.5$;
    $normale \leftarrow normale + \|n0 + n1\|$;
else
    $tang \leftarrow p1 - p0$;
    $tangLength \leftarrow |tang|$;
    $tang \leftarrow \|tang\|$;
    $t0 \leftarrow (n0 \times tang) \times n0$;
    $t1 \leftarrow (n1 \times -tang) \times n1$;
    $p2 \leftarrow tocke[index1] + t0 * tangLength$;
    $p3 \leftarrow tocke[index0] + t1 * tangLength$;
    $nova \leftarrow$ `CubicBezier`$(0.5, p0, p2, p1, p3)$;
    $tocke \leftarrow tocke + nova$;
    $normale \leftarrow normale + \|n0 + n1\|$;
end
return;

Vsako novo točko smo izračunali s pomočjo dveh obstoječih točk, ki predstavljata rob. Če je bil skalarni produkt dveh normal roba enak 0, 1 ali -1, smo izračunali povprečje točk roba. V nasprotnem primeru smo s pomočjo kubične Bezierove krivulje izračunali odmik točke nad površino. S tem smo dosegli, da imajo nove točke majhen odmik v primeru normal, ki ne kažejo v isto smer. Torej, če so normale kazale narazen, smo dobili izbočeno točko. Če so normale kazale skupaj, smo dobili vbočeno točko.

Za omenjeni izračun s pomočjo kubične bezierove krivulje smo morali izračunati štiri



Slika 4.7: Zamik nove točke glede na kubično bezierovo krivuljo.



(a) (b)

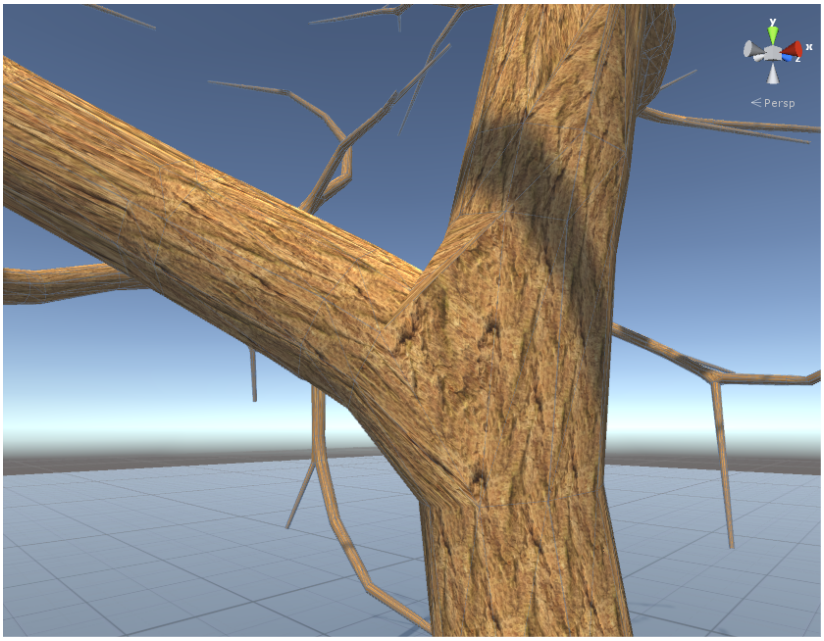
Slika 4.8: (a) Izvorna poligonska mreža; (b) Po deljenju mreže.

točke. Na sliki 4.7 sta točki P_1 in P_2 začetek in konec krivulje, drugi dve C_1 ter C_2 pa kontrolni točki. Kontrolni točki kontrolirata ukrivljenost krivulje. Prvi dve smo vzeli direktno z začetka in konca roba. Kontrolne točke pa smo izračunali s pomočjo križnega produkta tangente roba t_3 in normale trikotnika (n_1 in n_2), dobljeni vektor pa smo še enkrat uporabili za križni produkt, za pravo tangento kontrolne točke. Tangento t_1 smo

množili s konstanto med 0 in 1, da smo jo umetno skrajšali. Nato smo jo prišteli k prvi položajni točki. Tako smo dobili prvo kontrolno točko. Za drugo kontrolno točko smo izvedli isti postopek, le da smo za osnovno tangento roba t_3 uporabili negativno tangento.

Razliko v resoluciji poligonske mreže lahko opazimo med levo (Slika 4.8a) in desno sliko (Slika 4.8b).

4.1.6 Generiranje UV-koordinat



Slika 4.9: Teksturiran del drevesa.

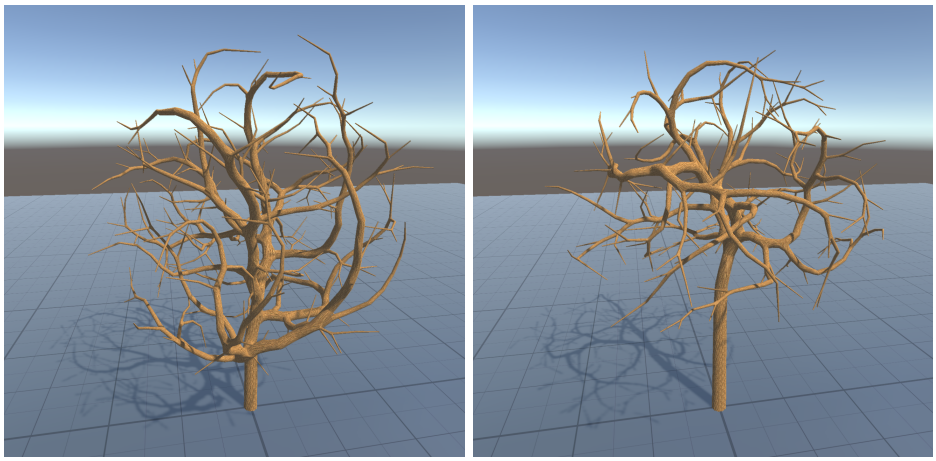
Koordinate smo izračunali na preprost način. Komponento x koordinate smo izračunali s pomočjo radialnega zamika okoli osi cilindra. Izračun komponente y koordinate je bil nekoliko bolj zapleten. Za ta namen smo vzeli odmik od korena drevesa do posamezne točke cilindra. Ta odmik smo nato projicirali na normalo odseka. Tako smo dobili y koordinato, ki se oddaljuje od središča drevesa. Pritrjevanje koordinat smo preprečili z uporabo funkcije Unity 3D Mathf.PingPong. Mathf.PingPong ves čas drži vrednost znotraj podanega intervala, ne glede na velikost vrednosti. Vrednost se povečuje in zmanjšuje linearno. Izmenjujoče ponavljanje tekstur vidimo na sliki 4.9.

4.2 Rezultati implementiranega generatorja

Vseh pet dreves v primerih uporablja iste parametre. Spreminja se le nabor vhodnih točk, ki spreminja obliko krošnje. Za drevesa na sliki 4.10a, 4.10b, 4.11a in 4.11b smo uporabili naslednje parametre:

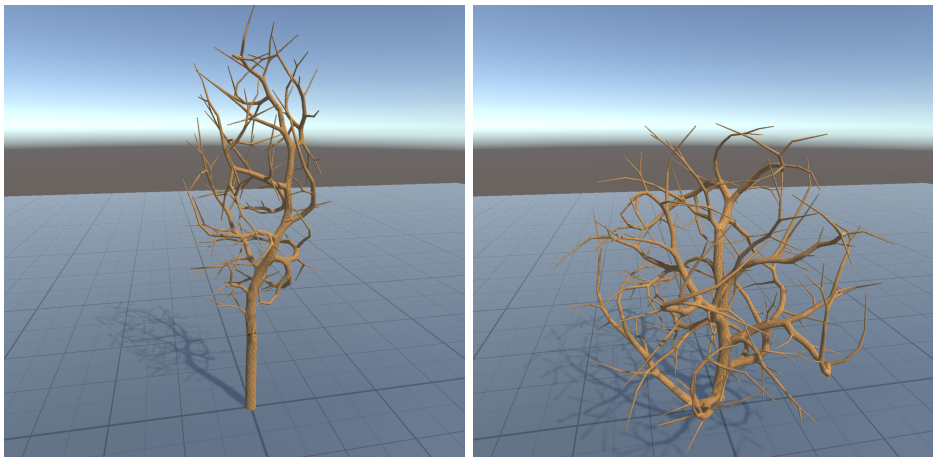
- minimalni kot med sorojenci: 45 stopinj,
- maksimalno število sorojencev: 2,
- polmer zunanjega območja: 1.5m,
- polmer notranjega območja: 0.47m,
- dolžina odsekov vej: 0.6m,
- maksimalna višina debla: 4.85m,
- dolžina odsekov debla: 1m.

Položaj množice točk je bil izračunan popolnoma naključno znotraj krogle. Na sliki 4.10a smo kroglo točk polmera 5m premaknili 7m visoko. Na sliki 4.10b smo enaki krogli odstranili spodnjo polovico. Na sliki 4.11a smo kroglo s prve slike stisnili po x in z koordinatah za 54%. Na sliki 4.11b smo osnovno kroglo premaknili tako, da se je dotikala tal. Na sliki 4.12 smo uporabili dve enaki krogli točk, ki sta se stikali v sredini, vendar pa smo morali zaradi števila trikotnikov in točk (16 bitna omejitev velikosti indeksov v pogonu Unity 3D) trikotnikov in točk izklopiti deljenje poligonske mreže. Na sliki 4.13 so drevesa, ki so šla skozi vse korake in so tako končni rezultat generatorja proceduralnih dreves.



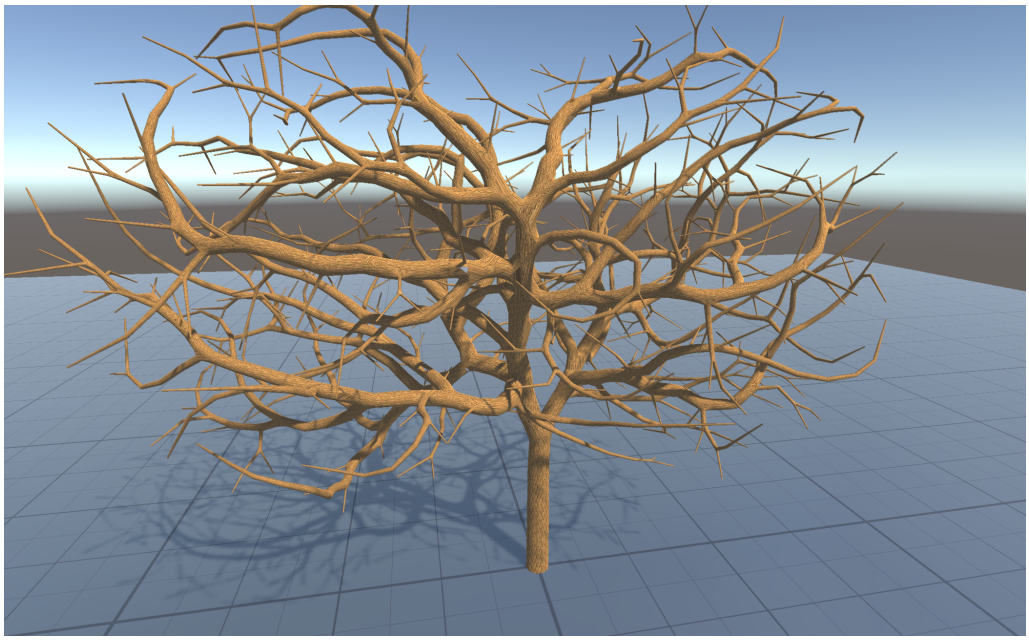
(a) (b)

Slika 4.10: (a) *Krogla točk*: drevo z 48999 točk in 94760 trikotnikov. (b) *Polkrogla točk*: drevo z 38276 točk in 73984 trikotnikov.

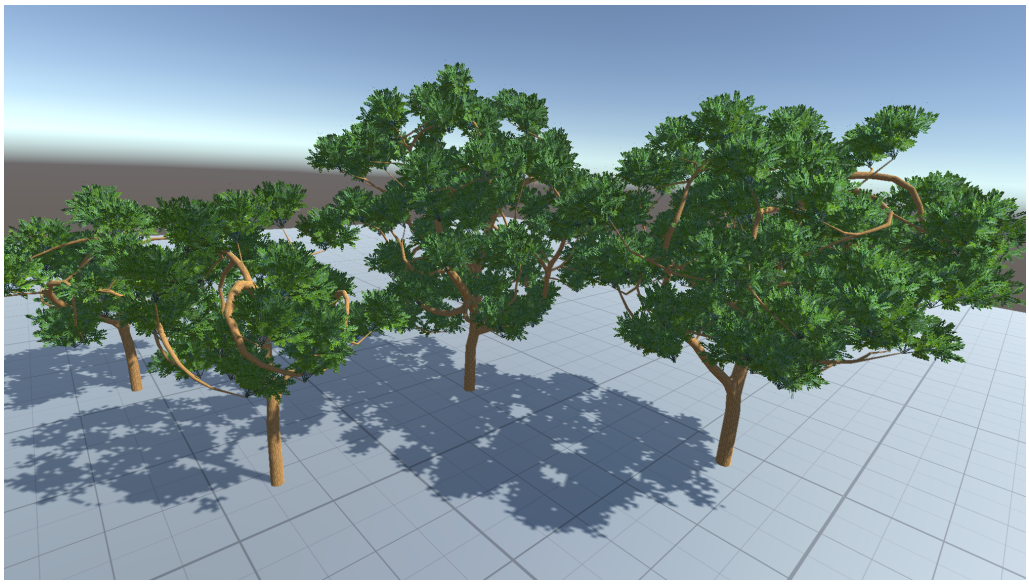


(a) (b)

Slika 4.11: (a) *Stisnjena krogla točk*: drevo z 25948 točk in 49896 trikotnikov. (b) *Krogla točk pri izhodišču*: grm z 48739 točk in 94248 trikotnikov.



Slika 4.12: *Dve sekajoči krogli*: drevo z 31380 točk in 58942 trikotnikov. Brez deljenja poligonske mreže.



Slika 4.13: Končana drevesa.

5 Sklepne ugotovitve

Ustvarili smo generator dreves, ki se je sposoben prilagajati vhodni množici točk in generirati popolnoma samostojno drevo. Tako generirana drevesa je mogoče uporabiti za statično gradnjo gozdov v scenah iger. To pomeni, da poligonske mreže dreves predgeneriramo v datoteke in jih nato zapečemo v sceno. Možna je tudi uporaba generatorja za generiranje dinamičnih dreves. Seveda bi za optimalno delovanje 3D-igre v realnem času morali ustvariti le nekaj različnih dreves in jih tako podvojiti čez celoten gozd. Podvajanje poligonskih mrež dreves bi pomagalo tudi pri nalagalnih časih in hitrosti izrisovanja. Za optimalno delovanje v igrah bi morali generirati več različnih nivojev natančnosti (angl.: Level of detail) poligonske mreže, saj v igrah v veliko primerih ni potrebe za izrisovanje visokoresolucijskih poligonskih mrež, oddaljenih od kamere oz. pogleda.

Veje in debla zgrajenih dreves v našem primeru so včasih videti nekoliko nerealistično. Še posebej je to opaziti v primerih, ko parametri generatorja niso pravilno nastavljeni, zato je za optimalno delovanje generatorja potrebnih nekaj ročnih nastavitev.

Generiranje celotnega drevesa je trajalo od pol sekunde do dveh sekund (rezultat smo dobili na procesorju Intel Core i7-2670QM CPU pri 2.20-GHz taktu). Tretjino časa je

generator porabil za gradnjo skeleta drevesa, preostali čas pa za sestavljanje poligonske mreže drevesa. Čas je bil odvisen od števila in gostote točk vhodne množice. Če smo želeli ustvariti drevo s krošnjo, je generator potreboval na istem procesorju približno sto dodatnih milisekund za gradnjo krošnje. Generator krošnje drevesa je ustvaril od tri do šestdeset tisoč poligonov. Generator poligonske mreže debla in vej pa od dva do dvajset tisoč poligonov. Ker je bilo drevo sestavljeno iz dveh posameznih delov, je grafični pogon moral narediti vsaj dva izrisa, in sicer a) izris vej in debla ter b) izris krošnje.

Generirana drevesa so zaradi ogromnega števila poligonov draga za izris v zahtevnejših igrah, kjer ni potrebe po podrobnih dreves. Izrisovanje dreves smo testirali v Unity3D pri najbolj podrobnih grafičnih nastavitvah (vključene ostre in mehke sence, glajenje robov ter osvetlitev iz usmerjene luči). Izrisovanje smo testirali na grafični kartici EVGA GTX 980Ti FTW. Izrisali smo lahko približno dvajset dreves preden se je izrisovanje upočasnilo pod trideset slik na sekundo oz. na približno trideset milisekund. Krivec za upočasnitev so seveda sence. To je zato, ker izrisovanje senc na prosojnih teksturah zahteva zahtevnejše operacije. Z izklopljenimi sencami smo zmanjšali porabljen čas izrisa na četrtino milisekunde. Gozdu smo nato lahko dodali trideset dreves. Tako da smo na koncu lahko izrisovali petdeset različnih dreves v realnem času. Kot smo že omenili bi za optimalno izrisovanje potrebovali več nivojev natančnosti. Poleg tega bi lahko poligonski mreži združili odvečne poligone ter zmanjšali gostoto pahljač z listi.

Ker smo generator napisali v programskem jeziku C# in ga izvajali s pomočjo tolmača Mono, je hitrost generiranja nekoliko počasnejša, kot če bi izvajali generator s pomočjo nizkonivojskih jezikov. Prostora za optimizacijo je veliko. Generiranje poligonske mreže bi bilo mogoče popolnoma paralelizirati v nizkonivojskih jezikih, kot sta C in C++. Generiranje celotnega drevesa je mogoče popolnoma izvesti tudi na jedrih OpenCL ali CUDA, kar bi nam omogočilo generiranje dreves v realnem času.

Za izpopolnitev in dovršitev dokončnega izdelka proceduralnega generatorja dreves bi bilo smiselno dodati še generator za teksture listov in cvetov. Tako bi se lahko prilagajali letnim časom, podnebnim značilnostim in mogoče odprti domišljiji. Poleg tega bi lahko v podrobnosti dodali še animacijo vetra, odpadanja listja, rast, odmiranje ipd. Za interaktivno izkušnjo bi lahko drevesu dodali fiziko in tako omogočili odziv drevesa glede na akcije igralca.

Literatura

[1] S. Oberbauer, B. Strain, Photosynthesis and successional status of Costa Rican rain forest trees, Photosynthesis Research 5 (3) (1984) 227–232.

[2] Z. Lam, S. A. King, Simulating tree growth based on internal and environmental factors, in: Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, GRAPHITE '05, ACM, New York, NY, USA, 2005, pp. 99–107.
url: http://doi.acm.org/10.1145/1101389.1101406

[3] C. Colditz, L. Coconu, O. Deussen, C. Hege, Realtime rendering of complex photorealistic landscapes using hybrid level-of-detail approaches, in: Trends in Real-Time Landscape Visualization and Participation, 2005, pp. 97–106.

[4] R. Habel, Real-time Rendering and Animation of Vegetation, PhD thesis, Vienna University of Technology, Vienna, Austria, 2009.
url: https://goo.gl/qyh7ka

[5] P. Decaudin, H. W. Jensen, A. K. (editors, F. Neyret), Rendering forest scenes in real-time, Eurographics Association, Norrköping, Sweden, 2004.

[6] J. Weber, J. Penn, Creation and rendering of realistic trees, in: Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '95, ACM, New York, NY, USA, 1995, pp. 119–128.
url: http://doi.acm.org/10.1145/218380.218427

[7] O. Argudo, A. Chica, C. Andujar, Single-picture reconstruction and rendering of trees for plausible vegetation synthesis, Computers & Graphics 57 (2016) 55 – 67.

[8] A. Runions, Modeling biological patterns using the space colonization algorithm,

MSc thesis, University of Calgary, Calgary, Alberta, CA (2008).
url: http://goo.gl/ffnDEd

[9] L. Xu, D. Mould, A procedural method for irregular tree models, Computers & Graphics 36 (8) (2012) 1036 – 1047.

[10] J. Lozej, Generiranje svetlobi prilagojenih dreves z uporabo genetskih algoritmov, diplomsko delo, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2015.

[11] R. Kogovšek, Tehnike proceduralnega modeliranja v računalniški grafiki, diplomsko delo, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2014.

[12] A. Jakulin, Interactive Vegetation Rendering with Slicing and Blending, in: Eurographics 2000, 20-25 August 2000, Interlaken, Switzerland.

[13] C. B. Barber, D. P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (1996) 469–483.

[14] M. Lysenko, Realtime constructive solid geometry, in: ACM SIGGRAPH 2007 Posters, SIGGRAPH '07, ACM, New York, NY, USA, 2007.
url: http://doi.acm.org/10.1145/1280720.1280864

[15] R. Pieterse, Procedural tree generation: Modeling branching structures as subdivision surfaces, MSc thesis, University of Cape Town (2012).
url: http://goo.gl/kUkY21

[16] D. Zorin, Subdivision on arbitrary meshes: Algorithms and theory, New York University, 2005.

A Osnovne geometrične metode

A.1 Projekcija točke na ravnino

Projekcija točke $\overrightarrow{r_R}$ na ravnino je definirana kot skalarni produkt vektorja $\overrightarrow{r_P}$ in normale $\overrightarrow{n}$. Rezultat skalarnega produkta predstavlja razdaljo točke $\overrightarrow{r_P}$ od ravnine. Skalarni produkt množimo z normalo $\overrightarrow{n}$, tako dobimo odmik. Odmik nato odštejemo od točke $\overrightarrow{r_P}$. Rezultat je projicirana točka $\overrightarrow{r_R}$.

$$\overrightarrow{r_R} = \overrightarrow{r_P} - [\overrightarrow{r_P} \cdot \overrightarrow{n}] * \overrightarrow{n} \tag{A.1}$$

A.2 Normala trikotnika

Normala $\overrightarrow{n}$ trikotnika je definirana kot križni produkt daljice $\overline{BA}$ in $\overline{CA}$. Rezultat križnega produkta moramo normalizirati.

$$\overrightarrow{r_P} = (\overrightarrow{r_B} - \overrightarrow{r_A}) \times (\overrightarrow{r_C} - \overrightarrow{r_A}) \tag{A.2}$$
$$\overrightarrow{n} = \frac{\overrightarrow{r_P}}{|\overrightarrow{r_P}|}$$

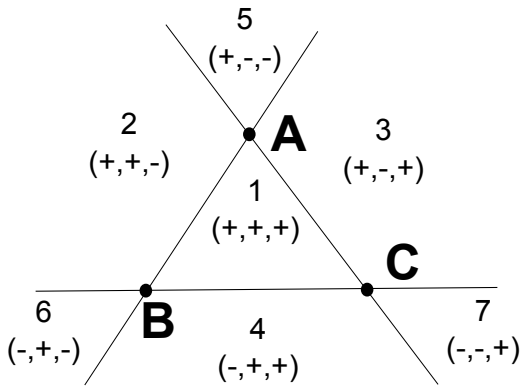
A.3 Najbližja točka na daljici

Najbližja točka $\overrightarrow{r_P}$ na daljici je definirana kot odmik od začetne točke $\overrightarrow{r_A}$, skalarnega produkta med normalo daljice in daljico $\overline{PA}$. Skalarni produkt je množen z normalo in v intervalu $[0, d]$. d je dolžina daljice $\overline{AB}$.

$$\begin{aligned} d &= |\overrightarrow{r_B} - \overrightarrow{r_A}| \\ \overrightarrow{n} &= \frac{\overrightarrow{r_B} - \overrightarrow{r_A}}{d} \\ t &= (\overrightarrow{r_P} - \overrightarrow{r_A}) \cdot \overrightarrow{n} \\ \overrightarrow{r_R} &= \overrightarrow{r_A} + \overrightarrow{n} * \max(\min(t, d), 0) \end{aligned} \tag{A.3}$$

A.4 Baricentrične koordinate trikotnika

Baricentrične koordinate trikotnika so definirane kot masa treh točk, ki so v ogliščih trikotnika. Baricentrične koordinate se uporabljajo za testiranje lokacij točk na trikotniku. Če je točka $\overrightarrow{r_P}$ znotraj trikotnika $\triangle ABC$, bodo vse tri koordinate (v, u, w) pozitivne. Iz preproste skice A.1 lahko izvemo, da imajo baricentrične koordinate 7 različnih področij. Vsako področje je definirano z drugačno kombinacijo predznakov.



Slika A.1: 7 področij baricentričnih koordinat.

$$\begin{aligned}
\overrightarrow{a_{V0}} = \overrightarrow{r_B} - \overrightarrow{r_A};\ \overrightarrow{a_{V1}} = \overrightarrow{r_C} - \overrightarrow{r_A};\ \overrightarrow{a_{V2}} = \overrightarrow{r_P} - \overrightarrow{r_A} \\
d_{00} = \overrightarrow{a_{V0}} \cdot \overrightarrow{a_{V0}};\ d_{01} = \overrightarrow{a_{V0}} \cdot \overrightarrow{a_{V1}};\ d_{11} = \overrightarrow{a_{V1}} \cdot \overrightarrow{a_{V1}} \\
d_{20} = \overrightarrow{a_{V2}} \cdot \overrightarrow{a_{V0}};\ d_{21} = \overrightarrow{a_{V2}} \cdot \overrightarrow{a_{V1}} \\
g = \frac{1}{d_{00} * d_{11} - d_{01} * d_{01}} \\
v = (d_{11} * d_{20} - d_{01} * d_{21}) * g \\
w = (d_{00} * d_{21} - d_{01} * d_{20}) * g \\
u = 1 - v - w
\end{aligned} \tag{A.4}$$

A.5 Najbližja točka na trikotniku

Najbližjo točko $\overrightarrow{p_R}$ na trikotniku smo iskali tako, da smo izračunali normalo trikotnika $\triangle ABC$. Nato projicirali točko $\overrightarrow{r_P}$ na ravnino trikotnika $\triangle ABC$. Nato smo izračunali baricentrične koordinate A.4. Kot je splošno znano, imajo baricentrične koordinate 7 možnih področij. Da najdemo najbližjo točko, moramo testirati predznake za 7 različnih možnosti.

- Vse tri vrednosti (v, u, w) so pozitivne. Najbližja točka je na trikotniku. Rezultat smo dobili s projekcijo točke $\overrightarrow{r_P}$ na ravnino trikotnika $\triangle ABC$.
- le vrednost v je negativna. Poiskali smo najbližjo točko na daljici $\overline{BC}$.
- le vrednost u je negativna. Poiskali smo najbližjo točko na daljici $\overline{AC}$.
- le vrednost w je negativna. Poiskali smo najbližjo točko na daljici $\overline{AB}$.
- vrednosti v in w sta negativni. Najbližja točka je bila $\overrightarrow{A}$.
- vrednosti u in w sta negativni. Najbližja točka je bila $\overrightarrow{B}$.
- vrednosti u in v sta negativni. Najbližja točka je bila $\overrightarrow{C}$.

Tako smo pokrili vse možne primere.

A.6 Izračun pravokotnega vektorja

Pravokotni vektor smo izračunali s pomočjo križnega produkta smeri in konstantnega vektorja, ki kaže v našem primeru levo (x komponenta je 1 ostale 0). V primeru, ko je bil

rezultat operacije vektor, katerega dolžina je bila manjša od 10^{-6}, smo izračun ponovili z drugim vektorjem, ki je kazal gor (y komponenta je 1, ostale so 0). Tako smo dobili enega izmed neskončno možnih pravokotnih vektorjev.